[0084] In one implementation, as described above, the contents of a device manifest file are derived from the pkd and processed crf file (called a psf file **280** once processed) information, as well as from the build manifest file (e.g., for the file list and attribute information). The crf file describes component-level dependencies, and is processed into a form that describes package-level dependencies.

[0085] Further, component settings (configuration information) can shadow one-another (in other words, there is a precedence ordering in events where two related settings exist on the system such that only one will win). Dependency and shadowing information at the component level are converted into package-level relations by way of a shadow order tool.

[0086] The shadow order tool produces a package shadow file (packagename.psf) for each package which shadows other packages according to the component relationships file **226** (**FIG. 2**). The .psf file for a package lists (one per line) the GUIDs of shadowed packages. The shadow order tool's inputs are the merged component to package mapping file (.cpm), the merged component relationships file (.crf) and the merged package definition file (.pkd). The output is a text file, including formatted text lines of the form:

[0087] <shadowed pkg GUID>, <shadowed pkg name>, <rule> (where rule=SHADOWS or DEPENDSON).

[0088] An example line is shown below:

[0089] 273ce4bf-d4ef-4771-b2ce-6fe2fa2b2666, SMARTFON, SHADOWS

[0090] From this information, the package generator creates a device manifest file, as shown in **FIG. 2** by a Device Manifest File Creation Process **262** and as generally described in **FIGS. 10A and 10B**. To this end, each GUID in the .psf file is added to the device manifest file, as generally represented via step **1010** of **FIG. 10A**. Note that in one implemetation, this includes determining if the GUID is a dependency GUID or a shadow GUID, and calling the appropriate add function on the device manifest object. When the GUIDS have been added, the device manifest object writes the device manifest file to the temporary directory, as generally represented via step **1030** of FIG. lOB.

[0091] As described in **FIG. 11**, the package file **232** is created, including the processed files described in the directive file, the rgu file(s) **214**, the xml file(s) **218**, and the device manifest file **260**, along with other package contents **282**, using a package file creation process such as a CAB file API (CABAPI). The CABAPI is intended to provide access to the contents of Cabinet files that are used as the transport mechanism for the files involved in an image update as part of the image update process.

[0092] To create the package, a PackageDefinition class is responsible for managing the overall creation of a Package. As part of the creation process, the PackageDefinition object creates a new subdirectory under the directory specified by the '_FLATRELEASEDIR' environment variable. The directory name is the package name, with the string "_PACKAGE_FILES" appended thereto, e.g., given a package named—"LANG"—, a directory named "LANG-_PACKAGE_FILES" would be created. This directory is created when the method SetDirectoryBase is called on the object. When the PackageDefinition object creates a package, the name of the resulting package file is the friendly name of the package with a ".PKG" extension. The package file conforms to the Microsoft CAB file specification for CAB version 1.3.

[0093] The PackagDefinition class provides the following public methods:

[0094] PackageDefinition(XipPackage pkg)—Constructor to create a package based on an XipPackage object.

[0095] SetDirectoryBase(string path)—Creates a new subdirectory under the specified directory.

[0096] Validate()—Determines if the package has the two required fields, a name and a GUID.

[0097] ReadManifesto—Causes the BuildManifest associated with this package to parse the appropriate build manifest file.

[0098] MakePackage()—Creates the actual package file for this package definition.

[0099] Further, when working with the various files for inclusion in the packages, it should be noted that executable code and data may be arranged in separate files. One significant advantage of is to facilitate a multi-language system, where the language-specific parts of a feature are placed into separate packages that are language specific. This creates a system where the packages that contain the executable code of the system are separate from the packages that contain language-specific components of the system. As a result, a patch for the executable code of a feature may be applied to any device, independent of any combination of languages that are installed on a that device.

[0100] More particularly, by construction, when a feature is built, the executable code (and language-independent data) are separated into one set of files, and the language-dependent data (and potentially code) into another set of files. These files are tagged as being part of the feature, but the language-dependent data files are further tagged being as language dependent. The system then moves those files into a separate package (e.g., described by a LocBuddy tag in the pkd file).

[0101] By way of example, a telephone-based feature may be in a library (e.g., tpcutil.dll) that is language-independent. The language-dependent resources for the phone feature are built into another resource dll, e.g., named tapres.dll, which is further localized for each language, e.g., it becomes tapres.dll.0409.mui (for US English), tapres.dll.0407.mui (for German), and so forth. These files are tagged as being part of the phone feature, but the language-specific files are further tagged as being language-specific, with suitable filenames. For example, filenames may be constructed by substituting suitable language tags into a location-based variable in the name, such as represented by tapres.dll.%LOCID%.mui. Then the file is processed for each language that is supported, and multiple LANGPHONE (the locbuddy) packages are generated, e.g., LANGPHONE_0409 (for US English), LANGPHONE_0407 (for German), and so forth. As a result, the system can later update the LANGPHONE region such as to fix bugs in